Evaluation of the CEC (Correct Eventual Consistency) Tool

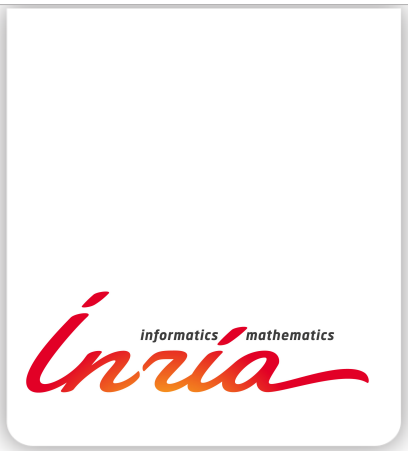
Sreeja S Nair

Evaluation of the CEC (Correct Eventual Consistency) Tool

Sreeja S. Nair

RESEARCH REPORT
N° 9111
October 2017
Project-Teams REGAL

ISSN 0249-6399 ISRN INRIA/RR--9111--FR+ENG



Evaluation of the CEC (Correct Eventual Consistency) Tool

Sreeja S. Nair

Project-Teams REGAL

Research Report n° 9111 — October 2017 — 27 pages

Abstract: Preserving invariants while designing distributed applications under weak consistency models is difficult. The CEC (*Correct Eventual Consistency Tool*) is meant to aid the application designer in this task. This report presents some specifications tried out using the tool and some recommendations for its improvement based on the usage experience.

Key-words: Consistency, Verification, Distributed applications

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Évaluation de l'outil CEC (Correct Eventual Consistency) pour la vérification des applications à cohérence à terme

Résumé : Le maintien des invariants, dans les applications réparties s'exécutant dans un modèle de cohérence faible, est un problème difficile. L'outil CEC (*Correct Eventual Consistency*) est destiné à aider le développeur d'application dans cette tâche. Notre rapport présente plusieurs exemples de spécifications vérifiées en utilisant cet outil, ainsi que quelques recommandations sur son usage, basées sur notre expérience d'utilisation.

Mots-clés : cohérence, vérification, applications distribuées

1 Introduction

The Correct Eventual Consistency (CEC) Tool is based on CISE logic developed by Gotsman et al.[1]. CISE logic provides reasoning on the correctness of a distributed application operating on top of a causally consistent database. According to the logic, a distributed application is guaranteed to uphold the invariant if each operation is sequentially correct, each operation is stable under the precondition of the other and the operations commute. The assumption of CISE logic is that the application is designed on top of a database layer which ensures causal delivery of updates.

The input to the tool is a specification written in the intermediate verification language, Boogie[2], with some annotations required by the tool. The tool performs verification checks on the specification and provides recommendations for synchronization. This report summarizes the evaluation of the tool in terms of ease of use.

We take an example of a simple decrement counter to illustrate the usage of the tool. First we show how to write a specification for a decrement counter normally used in sequential applications. We discuss the result given by the tool in terms of the synchronization recommendations. Then we will go to a better example by designing a counter with escrow. The tool gives some recommendations here as well. As a third step we specify a bounded counter CRDT and examine the result of the tool. We also show an example of a replicated growable array specification using the tool.

The report concludes with our recommendations to enhance the usability of the tool and a discussion on the possible future research and development directions for the tool.

2 Correct Eventual Consistency tool

Based on the CISE logic, a tool was developed and reported by Najafzadeh et al.[3]. As the authors have pointed out, the CISE tool is difficult to use since the user would need to study Z3 APIs in order to use it[4]. Marcelino et al.[5] has designed the CEC tool which works on the same principle and uses Boogie [2] with some annotations. The language except the annotations is specified by Leino[6]. The annotations are specific to this tool.

A specification consists of the following parts:

- Data structures and properties
- Variables
- Invariants
- Operations with pre and post conditions

The verification is done in two main steps :- specification correction and consistency verification. For each check, a boogie file is generated by the tool using the input specification. Boogie is invoked using these files and the output is examined to determine the status of the check.

The specification correction section checks whether

- each individual component of the specification is correct syntactically (*syntax check*)
- each operation satisfies the invariant individually (*safety check*)
- there is any contradictory clause in the specification (*anomaly check*)
- all variables which are modified by the operations have properly defined values after the execution (*completeness check*)

The analysis will proceed onto the consistency verification stage only if all the specification correction checks are passed. According to the event based proof rule described by Gotsman et al.[1], this stage ensures that the consistency of the specification upholds the application invariants. In a distributed application, each operation would be propagated from the replica which it originated from(*origin replica*) to all other replicas. That means, now the specificatin is sequentially correct at the origin replica.

The consistency verification section checks the pair of operations to see whether

- the pair of operations have opposing preconditions (*opposition check*)
- the precondition of one operation holds under the other operation (*stability check*)
- the pair of operations commute (*commutativity check*)

The ones that fail the opposition check do not need to be analysed further as they will never be executed concurrently. The second and third check in the consistency verification stage deals with the safety of concurrent operations. In case these checks fail, a token generation engine is invoked to generate concurrency tokens for ensuring safety. Currently, the token generator checks whether enforcing inequality on the input parameters would make the specification safe.

3 Examples

This section describes several examples tried using CEC tool. Each example is provided with a brief description on the context and the specification is explained. The complete sepcification of each example along with the tokens and the output from the tool is shown in the appendix. The specification of each example is examined in four parts:

- Data structures and properties
- Variables
- Invariants
- Operations

3.1 Decrement counter

Decrement counter is a count down counter with a lower bound. It is an integer which has `increment` and `decrement` operations. The specification can be examined as per the CEC tool input format.

- **Data structure and properties**
  The data structure is `int`, which is a native datatype supported by Boogie. A constant named `min` is defined to specify the lower bound which the counter must respect.
- **Variables**
  There is a single variable here, named `counter`, which is of type `int`.
- **Invariants**
  The only invariant is that the value of the `counter` should not be less than `min`.
- **Operations**
  The operations and the pre and post conditions are as follows:

– `increment`
This function is used to increment `counter` by `value` units. The only precondition for this operation is that the `value` by which the `counter` should be incremented should be positive.

– `decrement`
This function is used to decrease the value of `counter` by `value` units. The precondition for this operations is that `value` should be positive and the current value of the `counter` is more than `min` + `value`.

When we run the CEC tool with this specification, the tool indicates that for sequential applications it works perfectly fine. For the consistency tests, the tool shows that the precondition of `decrement` operation is not stable under the effect of another `decrement` operation.

For example, consider a scenario where `counter` was having a value of 5. The value of `min` is specified as 0. This means `counter` cannot decrease beyond 0. Now consider an operation `decrement(3)`. This operation when done in a sequential program satisfies the invariant. But if the same operation with the same parameter is executed at two different replicas, when the changes are propagated, the value of `counter` becomes -1, which violates the invariant. So once the operation is executed with the given parameter, a second operation is not possible, because it does not fulfill the precondition. This is the output of the tool. The tool also generates a token indicating that the `decrement` operation needs strong coordination. The format of the token is as shown below:

```
————————TOKEN MODEL————————
Tokens per operations:
decrement: decrement_all

Token conflicts:
decrement_all: decrement_all
```

This also means that the `increment` operation can be executed without any synchronization. In short, the tool points out the functions which would require synchronization and the application designer needs to implement it.

As we have seen, the `decrement` operation needs strong synchronization. So if a replica wants to perform a `decrement`, it should acquire a token which locks the counter for any `decrement` operation. We declare a global variable called `lock` of datatype `ReplicaID` which is to be acquired by any replica that needs to perform a `decrement` operation. We also specify the restriction stating that the `decrement` function should not originate from the same replica twice, which is based on the fact that causal delivery is ensured by the underlying database.

This leads to an application being unavailable to a user when another user needs to perform the `decrement` operation.

3.2 Decrement counter with escrow

In the counter described in the previous section, the locking happens for all `decrement` operations irrespective of the value of the counter and the value to be decremented. To overcome this drawback, the concept of escrow transactional model [7] is used to divide the rights for decrementing the counter between replicas. Najafzadeh [4] provides a specification for this counter which we reuse here. The specification is as follows:

- **Data structures and properties**
We have specified two datatypes here namely `Credit`, of type int, and `ReplicaID`.

- **Variables**
  The two variables used are `localCredit` for each replica and `globalCredit` which is a shared variable.
- **Invariants**
  The `localCredit` for all replicas and `globalCredit` should be always non-negative.
- **Operations** The four operations are defined as follows:
  - `increment`
    This operation increases the local credits of the local replica. The parameters to this operation are the `replica`, which is the `ReplicaID` of the local replica and `k`, the local credits to be added. The only precondition for the operation is that `k` should be positive.
  - `decrement`
    This operation is the inverse of the `increment` operation. It decreases the local credits of the local replica. The parameters to this operation are `replica` and `k`, the amount of local credits consumed. The preconditions for the operation is that `k` should be positive and the local replica should have sufficient local credits. The operation ensures that the local credits of `replica` is decreased by `k`.
  - `acquireCredit`
    This operation acquires some credit from the `globalCredit` and adds it to the `localCredit` of the replica. The parameters are similar to the previous operations. The precondition of the operation is that there should be sufficient `globalCredit` for the operation. The operation ensures `k` credits to be acquired by the local replica from `globalCredit`.
  - `releaseCredit`
    This operation releases the `localCredit` of `replica` to `globalCredit`. The parameters are the same as previous operations with the precondition that the `replica` has sufficient `localCredit`. The operation transfers credits from the `replica` to the `globalCredit`.

This specification is run against the CEC tool and the tool suggests that the precondition of the following pair of operations are not stable under another execution of the other.

- `acquireCredit` - `acquireCredit`
- `releaseCredit` - `releaseCredit`
- `releaseCredit` - `decrement`
- `decrement` - `decrement`

The tokens generated are as follows:

```
————————TOKEN MODEL————————
Tokens per operations:
decrement: decrement_param1
releaseCredit: releaseCredit_param1
acquireCredit: acquireCredit_all

Token conflicts:
decrement_param1: decrement_param1, releaseCredit_param1
acquireCredit_all: acquireCredit_all
releaseCredit_param1: decrement_param1, releaseCredit_param1
```

The restrictions which should be imposed here are that none of the pairs originate from the same replica. This makes sense since the CISE logic assumes causal delivery of updates.

This model requires locking only when `acquireCredit` function is called. We implement a lock similar to the decrement counter in the previous section and add an extra precondition to the `acquireCredit` function that the lock must be acquired. The tool is now run against the modified specification along with the restrictions and it shows a failure of the pair opposition test for a pair of `acquireCredit` operations. This makes sense as two `acquireCredit` operations originating from different replicas cannot run concurrently anymore. This design makes the application more available as the counter can operate seamlessly until each replica has enough local rights.

3.3 Decrement counter CRDT

The bounded counter[8] is a conflict-free replicated datatype (CRDT)[9] which helps to ensure numerical invariants. It is similar to the counter with escrow, but has a different implementation which takes the synchronization off the critical path. Balegas et al.[8] specify the design of a state-based CRDT. Instead the specification which we propose below is written as an operation-based CRDT. The specification is for a counter which has a lower bound.

- **Data structures and properties**
  The two datatypes defined are `matrix2d` and `matrix1d` for 2-dimensional and 1-dimensional matrices respectively. `replicas` indicate the number of replicas involved and `min` is the minimum value the counter should maintain. The matrices `R` (storing the information of the rights) and `U` (storing usage information) have dimensions `replicas` × `replicas` and `replicas` × 1 respectively. `value` and `local_rights` are computed as per the specification explained by Balegas et al.[8].

- **Invariants**
  The value of the counter should be always greater than or equal to the `min` value specified.

- **Operations with pre and post conditions**
  The three effector functions defined are as follows:

  – `increment`
  Increment operation takes two parameters

  * `id` - the origin replica
  * `n` - the new value of `R[id][id]` at the origin replica

  The prerequisite for the operation is simple, it has to be in a participating replica. The operation will edit the entry of `R[id][id]` in all the other replicas. The new value will be the sum of `n` and the existing value of `R[id][id]` .

  – `decrement`
  Decrement operation takes the same parameters as the `increment` operation with the same prerequisites. The operation edits the entry of `U[id]` , replacing it with the sum of `n` and the existing value of `U[id]` .

  – `transfer`
  Transfer operation is used to transfer credit between replicas. The parameters are:

  * `from` - the replica from which the credit is transferred
  * `to` - the replica to which the credit is transferred

* `n` - the new value of the entry `R[from][to]` at the origin replica

The prerequisite for the operation is that the replica cannot transfer credit to itself and `from` and `to` are participating replicas. The operation updates the entry at `R[from][to]` with the sum of `n` and the existing value of `R[from][to]` .

This specification is run against the CEC tool. The tool shows precondition instability of `decrement` operation with both `decrement` operation and `transfer` operation. The tool also generates the below token which shows that the operation pairs fail when they originate from the same replica.

```
————————TOKEN MODEL————————
Tokens per operations:
decrement: decrement_param1
transfer: transfer_param1

Token conflicts:
decrement_param1: decrement_param1, transfer_param1
transfer_param1: decrement_param1
```

Since the CISE logic assumes causal delivery in the underlying layer of the application specification, we can impose restrictions on the pair of operations to be generated from different replicas.

3.4 Improved Decrement counter CRDT

A slight modification of the above counter can result in a design which does not require explicit specification of causal delivery. We emulate the merge function provided in the state-based specification of the bounded counter[8]: instead of performing the addition operations on the elements of the matrix, we substitute them by the maximum of the current value and the incoming value. The parameter of the operation is the resultant of the computation performed in the local replica.

3.5 Replicated Growable Array

Replicated Growable Array(RGA) is a CRDT used in the context of collaborative editing [10]. It proposes a replicated data type for a linked list. Previous works have formally specified RGA and a recent work proved that RGA is eventually consistent [11]. The proof is based on modeling a network and specifying RGA on top of the model to show that given the network model, RGA eventually converges. Instead, we use the CISE logic to prove RGA correct.

- **Data structures and properties** `ListType` represents a map of nodes. A node is represented by a type `Ref` . The fields of a node are:
  - `content` , which contains the value of the node of type `Object`
  - `next` , which points to the next node
  - `clock_ins` , which is used to store the timestamp for insert operations
  - `clock_up` , which stores the timestamp for update/delete operations

  A variable is initiated, called `list` , which is of the data type `ListType` . `presence` is a variable that helps in tracking the nodes added while using insert operations. Its type is `[Ref]bool` .

  We define a set of additional objects as follows:

- `tombstone` of type `Object`, indicating that the object is deleted
- `head` of type `Ref`, indicating the start of the list
- `tail` of type `Ref`, indicating the end of the list

For each of these fields and special objects, we need to specify the properties. The property of `presence` flag requires that if a node is present in the list, the next node which it points to must also be present in the list. This ensures that a node already inserted does not point to a node yet to be inserted to the list. For the `head` and `tail` nodes, we define the properties as below:

- `clock_ins` and `clock_up` are set to 0
- `content` is set to a tombstone (indicating nothing)
- `presence` to true to show that they will be present in the list

For all nodes except the `head` and `tail`, the `clock_ins` and `clock_up` should have non-zero values. The `clock_ins` and `clock_up` are meant to have information about the origin replica and the time of the operation. Hence it is guaranteed that no two `clock_ins` or `clock_up` would be equal.

The support functions and their functionalities are describes as follows:

- `create_node` creates a new node and returns it. The created node will neither be a head nor the tail and the function will never return the same object if either the content or the timestamp of the object differs.
- `get_actual_reference` returns the exact location where the insert operation can insert a new node. The node to be inserted is to be placed to the right side of the base node but after the nodes which are inserted later at the same place. The nodes inserted later will have a greater value for the clock. Until it finds a node such that the next of the node has a clock value lesser than the current operation clock value, it goes on searching recursively.
- `is_successor` checks whether the parameters `successor` has a connection from the parameter `base`. This function is used to define the property of the list that the list should start from the head and end in the tail.

The property that no two nodes have the same next has been specified to avoid circles in the list and enforces the list to be linear.

- **Invariants**

  The invariants of a linked list are only two :

  - A `head` should be present
  - A `tail` should be present

- **Operations with pre and post conditions**

  The four operations are `insert`, `update`, `delete` and `read`. The following shows the pre and post conditions of each operation along with the variables it modifies.

  - `insert`

    Insert operation takes three parameters

  * `base` - the node to the right of which the new value is to be inserted
  * `new` - the value to be inserted
  * `time` - the number indicating the origin replica and the timestamp of the operation

  The preconditions for the operation are that:

  * `base` is not a `tail`
  * `base` is already present in the `list`
  * `new` is not a `tombstone`
  * the new node created using `new` and `time` is not the same as base and it is not already present in the list

  This operation modifies both the variables, `list` and `presence`. A new node is created using `new` and `time` parameters and it is inserted to the right of the node which is obtained by calling the `get_actual_reference` function. The `next` fields of the newly created node and the reference node to the right of which it is addded to are modified accordingly. For the new node created the `presence` flag is also set.

- `update`
  Update operation takes three parameters

  * `base` - the node to be updated
  * `obj` - the new value
  * `time` - the timestamp of the operation

  The preconditions for the operation are that `base` is not a `tail` and `base` is already present in the `list`. This operation modifies `list` by updating `content` of `base` node to `obj` and updating `clock_up`. This modification happens only when `clock_up` of the node is less than the operation timestamp, else it returns the same list without any modifications.

- `delete`
  Delete operation takes two parameters

  * `base` - the node to be deleted
  * `time` - the number indicating the origin replica and the timestamp of the operation

  The preconditions for the operation are the same as that of the `update` operation. The only difference between `update` and `delete` operations is that the `content` of the `base` node is updated to a `tombstone`.

- `read`
  Read operation takes only the node of which the value is to be returned, `base`, as the input. The preconditions are similar to `update` and `delete` and it does not have any impact on the list.

The tool is run with the specification input. The following are the issues pointed out by the tool:

- `insert` - `insert` stability issue
- `delete` - `update` commutativity issue
- `update` - `update` commutativity issue

The tokens generated by the solver is as follows:

```
----------TOKEN MODEL----------
Tokens per operations:
update: update_param3
insert: insert_param3
delete: delete_param2

Token conflicts:
delete_param2: update_param3
insert_param3: insert_param3
update_param3: delete_param2, update_param3
```

The tokens indicated as in the same notation as we specified earlier. We know that no two timestamps for a pair of operations would be equal. When we specify this restriction, the verification succeeds.

4 Conclusion

The Correct Eventual Consistency (CEC) tool was used to specify some sample replicated data structures. We showed how the tool captured the behaviours of different implementations of a decrement counter and also the replicated growable array.

From the experience gained from the tool usage, we suggest the following improvements to the tool (in decreasing order of the usefulness), categorized into sections:

- Debugging support

  As of now, the tool gives information on the result of the test - pass or fail, along with an error trace from boogie. For the user, who is not aware of the intermediate boogie files generated by the tool, the error traces are not useful. To support the debugging experience, we need to see how to provide useful information for each step of verification. The following are the suggestions for each step:

  - Specification correction
    - Syntax check
      Boogie check provides the exact position of the syntax issue in the input file. Since the tool is reordering the format of the specification before sending it to Boogie, it would be useful to remap the error position to the position in the specification given by the user. This information is quite sufficient for identifying the issue.
    - Safety check
      A counter example can help to understand more about the issue which violates the invariant. Further to that the invariant specified by the user can be broken down and fed to the Z3 solver to narrow down what exactly caused the violation. In some cases, strengthening the preconditions might be needed. The weakest precondition calculus can be used to derive the weakest precondition. The derived precondition needs to be a subset of the specified precondition. The precondition might need further strengthening in some cases though.
    - Anomaly check
      This is the trickiest check to debug. This test is designed in a way that the test fails if the boogie verification succeeds, thereby having no error trace and counter model generated. This leads to a situation where the user cannot get a counter example or any pointer to the failure. We have to devise an approach to find a good solution for debugging this test.

    - Completeness check
      Completeness check fails when the user doesn't specify all the components of the variables modified in the procedure. If we can point out exactly the list of components left out, it would be useful. The old and new values obtained from the counter model can be used here to point out the components whose behavior is to be specified.
  - Consistency verification
    - Opposition check
      This checks whether the preconditions are disjoint sets. This is modeled in a way similar to the anomaly check and hence it is difficult to have a support in debugging.
    - Stability check
      The debugging support for this step can be modeled as the same in the case of the safety check.
    - Commutativity check
      This step can borrow the ideas from the completeness check.

- Counter example comprehension

  Once the verification fails, the previous step will allow the user to identify the category of issue with the specification, like whether the individual safety test failed. Apart from this, a counter example will also help the user to understand more about the issue with the specification. For this, the feature of counter models which are generated by the SMT solve Z3, can be used. As a first level information, the parameters of the failed operation along with the values of the components in the invariant can be displayed. If the user wishes to know the state of the global variables before and after the execution of the operation, that can be provided as well. Boogie's "CaptureState" functionality can help here.

- Experimentation platform

  The idea is to host the tool as a service and collect usage logs. The platform can be used as part of some coursework for the students to experiment application and/or data structure specifications. The logs can be stored and examined offline. This will help in identifying the bottlenecks in specification writing. It will also be useful later in studying the patterns in specifications.

- Token generation

  - The current token generation part takes a huge amount of time to suggest tokens, especially when the number of parameters are more. This needs to be optimised. The counter example generated from the previous step might be useful here to narrow down the search space of the tokens.
  - The token generator only generates tokens for inequality of parameters. It can be made to handle other relations as well.
  - The token generator suggests tokens by restricting the parameters. When the parameters are structured data types, this leads to a coarse locking, even if only a particular component only needs to be taken into account. This can be extended to support token generation in a finer level, considering the components of the structured data types.

- Easier specification

  While writing the specification, the part which needed the most effort is the specification of properties, especially when the data types of the variables are complex data structures. If some support system is present to aid the programmers in guiding them to write the correct minimal properties, it would make the tool easier to use. Some possible approaches are:

  - Generate specification (or the axioms of the specification) from the code and/or test cases
  - Provide a set of libraries for the usual special objects defined in the properties, such as "null".
  - Identify different patterns for properties and provide reusable code snippets.

- Supporting state-based datatypes

  The current version of the tool supports the specification in the form of operation-based replicated data types. This can be extended to support the verification of state-based replicated data types as well. The extended version needs to check whether

  - Each operation including the `merge` is sequentially safe
  - The precondition of `merge` is stable under itself
  - `merge` is commutative

Acknowledgement

The author would like to thank Marc Shapiro and Paolo Viotti for their guidance and feedback and Gonçalo Marcelino for answering questions related to the tool. This research is supported in part by the RainbowFS project (*Agence Nationale de la Recherche*, France, number ANR-16-CE25-0013-01) and by European H2020 project 732 505 LightKone (2017–2020).

References

[1] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. "'Cause I'M Strong Enough: Reasoning About Consistency Choices in Distributed Systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, 2016, pp. 371–384.

[2] *Boogie*. 2017. URL: `https://github.com/boogie-org/boogie`.

[3] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. "The CISE Tool: Proving Weakly-consistent Applications Correct". In: *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*. PaPoC '16. London, United Kingdom: ACM, 2016, 2:1–2:3.

[4] Mahsa Najafzadeh. "The Analysis and Co-design of Weakly-Consistent Applications". Theses. Université Pierre et Marie Curie, Apr. 2016.

[5] Gonçalo Marcelino, Valter Balegas, and Carla Ferreira. "Bringing Hybrid Consistency Closer to Programmers". In: *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC'17. Belgrade, Serbia: ACM, 2017, 6:1–6:4.

[6] Rustan Leino. "This is Boogie 2". In: Microsoft Research, 2008. URL: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[7] Patrick E. O'Neil. "The Escrow Transactional Method". In: *ACM Trans. Database Syst.* 11.4 (Dec. 1986), pp. 405–430. ISSN: 0362-5915. DOI: 10.1145/7239.7265. URL: http://doi.acm.org/10.1145/7239.7265.

[8] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguiça. "Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants". In: *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. 2015, pp. 31–36.

[9] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-free Replicated Data Types". In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400.

[10] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. "Replicated Abstract Data Types: Building Blocks for Collaborative Applications". In: *J. Parallel Distrib. Comput.* 71.3 (Mar. 2011), pp. 354–368.

[11] Victor Gomes, Martin Kleppmann, Dominic Mulligan, and Alastair Beresfordn. "Verifying Strong Eventual Consistency in Distributed Systems". In: (2011). arXiv: 1707.01747 [cs.DC]. URL: https://arxiv.org/abs/1707.01747.

A Specification inputs to the tool and the corresponding output

A.1 Decrement Counter

Specification

```
@import;

@init
type ReplicaID;
const min:int;

@variables
var counter:int;
var lock:ReplicaID;

@invariant
counter >= min;

@operations
procedure increment(replica:ReplicaID, value:int)
modifies counter;
requires value > 0;
ensures counter == old(counter) + value;

procedure decrement(replica:ReplicaID, value:int)
modifies counter;
requires value > 0 && (counter >= value + min) && lock == replica;
ensures counter == old(counter) - value;
```

Token

```
[
    {
        "operation1": "decrement",
        "operation2": "decrement",
        "restrictions": ["@1 != @3"]
    }
]
```

Result

```
################ SPECIFICATION CORRECTION TESTS ################

----------------BASE VERIFICATION----------------
IMPORTS TEST: (PASSED)
INITIALIZER TEST: (PASSED)
VARIABLES TEST: (PASSED)
INVARIANTS TEST: (PASSED)
decrement OPERATION TEST: (PASSED)
increment OPERATION TEST: (PASSED)
FULL TEST: (PASSED)

----------------LOGIC VERIFICATION----------------
decrement SAFETY TEST: (PASSED)
increment SAFETY TEST: (PASSED)

decrement ABSURD TEST: (PASSED)
increment ABSURD TEST: (PASSED)
```

```
decrement COMPLETENESS TEST: (PASSED)
increment COMPLETENESS TEST: (PASSED)

----------------RESTRICTIONS VERIFICATION----------------

RESTRICTIONS WERE LOADED CORRECTLY

################ CONSISTENCY TESTS ################

----------------PAIR OPPOSITION VERIFICATION----------------
decrement decrement OPPOSITION TEST: (FAILED)
increment increment OPPOSITION TEST: (PASSED)
increment decrement OPPOSITION TEST: (PASSED)

----------------PAIR STABILITY VERIFICATION----------------
increment increment STABILITY TEST: (PASSED)
increment decrement STABILITY TEST: (PASSED)

----------------PAIR COMMUTATIVITY VERIFICATION----------------
increment increment COMMUTATIVITY TEST: (PASSED)
increment decrement COMMUTATIVITY TEST: (PASSED)

----------------EQUALITY SOLVER----------------
----------------TOKEN MODEL----------------
Tokens per operations:

Token conflicts:
```

A.2 Decrement Counter with escrow

Specification

```
@import;

@init
type Credit = int;
type ReplicaID;


@variables
var localCredit:[ReplicaID]Credit;
var globalCredit:Credit;
var l:ReplicaID;

@equals [ReplicaID]Credit @as forall r:ReplicaID :: @this[r] == @other[r];
@equals Credit @as @this == @other;

@invariant
(forall r:ReplicaID :: localCredit[r] >= 0) && (globalCredit >= 0);
@operations

procedure acquireCredit(replica:ReplicaID, k:Credit)
modifies globalCredit, localCredit;
requires k > 0 && globalCredit >= k && l == replica;
ensures (forall r:ReplicaID :: (r == replica ==> localCredit[r] == old(localCredit
    )[r] + k) && (r != replica ==> localCredit[r] == old(localCredit)[r])) &&
    globalCredit == old(globalCredit) - k;

procedure releaseCredit(replica:ReplicaID, k:Credit)
modifies globalCredit, localCredit;
requires k > 0 && localCredit[replica] >= k;
```

```
ensures (forall r:ReplicaID :: (r == replica ==> localCredit[r] == old(localCredit
    )[r] - k) && (r != replica ==> localCredit[r] == old(localCredit)[r])) &&
    globalCredit == old(globalCredit) + k;

procedure increment(replica:ReplicaID, k:Credit)
modifies localCredit;
requires k > 0;
ensures forall r:ReplicaID :: (r == replica ==> localCredit[r] == old(localCredit)
    [r] + k) && (r != replica ==> localCredit[r] == old(localCredit)[r]);

procedure decrement(replica:ReplicaID, k:Credit)
modifies localCredit;
requires k > 0 && localCredit[replica] >= k;
ensures forall r:ReplicaID :: (r == replica ==> localCredit[r] == old(localCredit)
    [r] - k) && (r != replica ==> localCredit[r] == old(localCredit)[r]);
```

Token

```
[
    {
        "operation1": "releaseCredit",
        "operation2": "releaseCredit",
        "restrictions": ["@1 != @3"]
    },
    {
        "operation1": "decrement",
        "operation2": "decrement",
        "restrictions": ["@1 != @3"]
    },
    {
        "operation1": "acquireCredit",
        "operation2": "acquireCredit",
        "restrictions": ["@1 != @3"]
    },
    {
        "operation1": "releaseCredit",
        "operation2": "decrement",
        "restrictions": ["@1 != @3"]
    }
]
```

Result

```
################ SPECIFICATION CORRECTION TESTS ################

----------------BASE VERIFICATION----------------
IMPORTS TEST: (PASSED)
INITIALIZER TEST: (PASSED)
VARIABLES TEST: (PASSED)
INVARIANTS TEST: (PASSED)
decrement OPERATION TEST: (PASSED)
acquireCredit OPERATION TEST: (PASSED)
increment OPERATION TEST: (PASSED)
releaseCredit OPERATION TEST: (PASSED)
FULL TEST: (PASSED)

----------------LOGIC VERIFICATION----------------
decrement SAFETY TEST: (PASSED)
acquireCredit SAFETY TEST: (PASSED)
increment SAFETY TEST: (PASSED)
releaseCredit SAFETY TEST: (PASSED)
```

```
decrement ABSURD TEST: (PASSED)
acquireCredit ABSURD TEST: (PASSED)
increment ABSURD TEST: (PASSED)
releaseCredit ABSURD TEST: (PASSED)

decrement COMPLETENESS TEST: (PASSED)
acquireCredit COMPLETENESS TEST: (PASSED)
increment COMPLETENESS TEST: (PASSED)
releaseCredit COMPLETENESS TEST: (PASSED)

----------RESTRICTIONS VERIFICATION----------

RESTRICTIONS WERE LOADED CORRECTLY

############### CONSISTENCY TESTS ###############

----------PAIR OPPOSITION VERIFICATION----------
releaseCredit releaseCredit OPPOSITION TEST: (PASSED)
decrement decrement OPPOSITION TEST: (PASSED)
acquireCredit acquireCredit OPPOSITION TEST: (FAILED)
releaseCredit decrement OPPOSITION TEST: (PASSED)
acquireCredit releaseCredit OPPOSITION TEST: (PASSED)
acquireCredit increment OPPOSITION TEST: (PASSED)
acquireCredit decrement OPPOSITION TEST: (PASSED)
releaseCredit increment OPPOSITION TEST: (PASSED)
increment increment OPPOSITION TEST: (PASSED)
increment decrement OPPOSITION TEST: (PASSED)

----------PAIR STABILITY VERIFICATION----------
releaseCredit releaseCredit STABILITY TEST: (PASSED)
decrement decrement STABILITY TEST: (PASSED)
releaseCredit decrement STABILITY TEST: (PASSED)
acquireCredit releaseCredit STABILITY TEST: (PASSED)
acquireCredit increment STABILITY TEST: (PASSED)
acquireCredit decrement STABILITY TEST: (PASSED)
releaseCredit increment STABILITY TEST: (PASSED)
increment increment STABILITY TEST: (PASSED)
increment decrement STABILITY TEST: (PASSED)

----------PAIR COMMUTATIVITY VERIFICATION----------
releaseCredit releaseCredit COMMUTATIVITY TEST: (PASSED)
decrement decrement COMMUTATIVITY TEST: (PASSED)
releaseCredit decrement COMMUTATIVITY TEST: (PASSED)
acquireCredit releaseCredit COMMUTATIVITY TEST: (PASSED)
acquireCredit increment COMMUTATIVITY TEST: (PASSED)
acquireCredit decrement COMMUTATIVITY TEST: (PASSED)
releaseCredit increment COMMUTATIVITY TEST: (PASSED)
increment increment COMMUTATIVITY TEST: (PASSED)
increment decrement COMMUTATIVITY TEST: (PASSED)

----------EQUALITY SOLVER----------
----------TOKEN MODEL----------
Tokens per operations:

Token conflicts:
```

A.3 Bounded Counter

Specification [1]

```
@import;
```

```
@init
type matrix2d=[int,int]int;
type matrix1d=[int]int;
const replicas:int;
const min:int;

axiom(forall r:int, U:matrix1d :: r >= 0  && r <= replicas ==> U[r] >= 0);
axiom(forall r1, r2:int, R:matrix2d :: r1 >= 0 && r1 <= replicas && r2 >= 0 && r2
    <= replicas ==> R[r1,r2] >= 0);
axiom(replicas > 0 && replicas < 99);

function value(R:matrix2d, U:matrix1d) returns(int)
{
  min + diagonal_sum(R, 0) + sum(U, 0)
}
function diagonal_sum(R:matrix2d, index:int) returns(int)
{
  if index < replicas then (R[index,index] + diagonal_sum(R, index + 1)) else (0)
}
function sum(U:matrix1d, index:int) returns(int)
{
  if index < replicas then (U[index] + sum(U, index + 1)) else (0)
}
function local_rights(id:int, R:matrix2d, U:matrix1d) returns(int)
{
  R[id,id] + column_sum(R, id, 0) - row_sum(R, id, 0) - U[id]
}
function column_sum(R:matrix2d, col:int, index:int) returns(int)
{
  if index > replicas then (R[index,col] + column_sum(R, col, index + 1)) else (0)
}
function row_sum(R:matrix2d, row:int, index:int) returns(int)
{
  if index > replicas then (R[row,index] + row_sum(R, row, index + 1)) else (0)
}

function increment_effect(R:matrix2d, oldR:matrix2d, id:int, n:int) returns(bool)
{
  (forall r1, r2:int :: ((r1 == id && r2 == id) ==> R[r1,r2] == oldR[r1,r2] + n)
     && ((r1 != id || r2 != id) ==> R[r1,r2] == oldR[r1,r2]))
}
function decrement_effect(U:matrix1d, oldU:matrix1d, id:int, n:int) returns(bool)
{
  (forall r:int :: (r == id ==> U[r] == oldU[r] + n) && (r != id ==> U[r] == oldU[
     r]))
}
function transfer_effect(R:matrix2d, oldR:matrix2d, from:int, to:int, n:int)
    returns(bool)
{
  (forall r1, r2:int :: ((r1 == from && r2 == to) ==> R[r1,r2] == oldR[r1,r2] + n)
     && ((r1 != from || r2 != to) ==> R[r1,r2] == oldR[r1,r2]))
}

@variables
var R:matrix2d;
var U:matrix1d;

@equals matrix2d @as forall r1, r2:int :: @this[r1,r2] == @other[r1,r2];
@equals matrix1d @as forall r:int :: @this[r] == @other[r];

@invariant
value(R, U) >= min;
```

```
@operations

procedure increment(id:int, n:int)
modifies R;
requires n > 0 && id >= 0 && id < replicas;
ensures increment_effect(R, old(R), id, n);

procedure decrement(id:int, n:int)
modifies U;
requires n > 0  && id >= 0 && id < replicas && local_rights(id, R, U) >= n;
ensures decrement_effect(U, old(U), id, n);

procedure transfer(from:int, to:int, n:int)
modifies R;
requires n > 0  && from >= 0 && from < replicas  && to >= 0 && to < replicas &&
    from != to && local_rights(from, R, U) >= n;
ensures transfer_effect(R, old(R), from, to, n);
```

Token

```
[
    {
        "operation1": "decrement",
        "operation2": "decrement",
        "restrictions": ["@1 != @3"]
    },
    {
        "operation1": "decrement",
        "operation2": "transfer",
        "restrictions": ["@1 != @3"]
    }
]
```

Result

```
################ SPECIFICATION CORRECTION TESTS ################

----------------BASE VERIFICATION----------------
IMPORTS TEST: (PASSED)
INITIALIZER TEST: (PASSED)
VARIABLES TEST: (PASSED)
INVARIANTS TEST: (PASSED)
increment OPERATION TEST: (PASSED)
transfer OPERATION TEST: (PASSED)
decrement OPERATION TEST: (PASSED)
FULL TEST: (PASSED)

----------------LOGIC VERIFICATION----------------
increment SAFETY TEST: (PASSED)
transfer SAFETY TEST: (PASSED)
decrement SAFETY TEST: (PASSED)

increment ABSURD TEST: (PASSED)
transfer ABSURD TEST: (PASSED)
decrement ABSURD TEST: (PASSED)

increment COMPLETENESS TEST: (PASSED)
transfer COMPLETENESS TEST: (PASSED)
decrement COMPLETENESS TEST: (PASSED)

----------------RESTRICTIONS VERIFICATION----------------
```

```
RESTRICTIONS WERE LOADED CORRECTLY

################ CONSISTENCY TESTS ################

----------------PAIR OPPOSITION VERIFICATION----------------
decrement decrement OPPOSITION TEST: (PASSED)
decrement transfer OPPOSITION TEST: (PASSED)
increment increment OPPOSITION TEST: (PASSED)
increment decrement OPPOSITION TEST: (PASSED)
increment transfer OPPOSITION TEST: (PASSED)
transfer transfer OPPOSITION TEST: (PASSED)

----------------PAIR STABILITY VERIFICATION----------------
decrement decrement STABILITY TEST: (PASSED)
decrement transfer STABILITY TEST: (PASSED)
increment increment STABILITY TEST: (PASSED)
increment decrement STABILITY TEST: (PASSED)
increment transfer STABILITY TEST: (PASSED)
transfer transfer STABILITY TEST: (PASSED)

----------------PAIR COMMUTATIVITY VERIFICATION----------------
decrement decrement COMMUTATIVITY TEST: (PASSED)
decrement transfer COMMUTATIVITY TEST: (PASSED)
increment increment COMMUTATIVITY TEST: (PASSED)
increment decrement COMMUTATIVITY TEST: (PASSED)
increment transfer COMMUTATIVITY TEST: (PASSED)
transfer transfer COMMUTATIVITY TEST: (PASSED)

----------------EQUALITY SOLVER----------------
----------------TOKEN MODEL----------------
Tokens per operations:

Token conflicts:
```

A.4 Modified Bounded Counter

Specification [1]

```
@import;

@init
type matrix2d=[int][int]int;
type matrix1d=[int]int;
const replicas:int;
const min:int;

axiom(forall r:int, U:matrix1d :: r >= 0  && r < replicas ==> U[r] >= 0);
axiom(forall r1, r2:int, R:matrix2d :: r1 >= 0 && r1 < replicas && r2 >= 0 && r2 <
      replicas ==> R[r1][r2] >= 0);
axiom(replicas > 0 && replicas < 99);

function value(R:matrix2d, U:matrix1d) returns(int)
{
  min + diagonal_sum(R, 0) + sum(U, 0)
}
function diagonal_sum(R:matrix2d, index:int) returns(int)
{
  if index < replicas then (R[index][index] + diagonal_sum(R, index + 1)) else (0)
```

[1] The replicas are bounded between 0 and 99, since Boogie gives a verification failure for replicas more than 99. This is yet to be investigated though.

```
}
function sum(U:matrix1d, index:int) returns(int)
{
  if index < replicas then (U[index] + sum(U, index + 1)) else (0)
}
function local_rights(id:int, R:matrix2d, U:matrix1d) returns(int)
{
  R[id][id] + column_sum(R, id, 0) - row_sum(R, id, 0) - U[id]
}
function column_sum(R:matrix2d, col:int, index:int) returns(int)
{
  if index > replicas then (R[index][col] + column_sum(R, col, index + 1)) else
      (0)
}
function row_sum(R:matrix2d, row:int, index:int) returns(int)
{
  if index > replicas then (R[row][index] + row_sum(R, row, index + 1)) else (0)
}

function edit_matrix2d(R:matrix2d, oldR:matrix2d, row:int, column:int, value:int)
    returns(bool)
{
  (forall r, c:int :: ((r == row && c == column) ==> R[r][c] == max(value, oldR[r
      ][c])) && ((r != row || c != column) ==> R[r][c] == oldR[r][c]))
}
function edit_matrix1d(U:matrix1d, oldU:matrix1d, id:int, value:int) returns(bool)
{
  (forall r:int :: (r == id ==> U[r] == max(value, oldU[r])) && (r != id ==> U[r]
      == oldU[r]))
}
function max(a:int, b:int) returns(int)
{
  (if a > b then a else b)
}

@variables
var R:matrix2d;
var U:matrix1d;

@equals matrix2d @as forall r1, r2:int :: @this[r1][r2] == @other[r1][r2];
@equals matrix1d @as forall r:int :: @this[r] == @other[r];

@invariant
value(R, U) >= min;

@operations

procedure increment(id:int, n:int)
modifies R;
requires id >= 0 && id < replicas;
ensures edit_matrix2d(R, old(R), id, id, n);

procedure decrement(id:int, n:int)
modifies U;
requires id >= 0 && id < replicas;
ensures edit_matrix1d(U, old(U), id, n);

procedure transfer(from:int, to:int, n:int)
modifies R;
requires from >= 0 && from < replicas  && to >= 0 && to < replicas && from != to;
ensures edit_matrix2d(R, old(R), from, to, n);
```

Token

There are no tokens required for this specification.

Result

```
################ SPECIFICATION CORRECTION TESTS ################

----------------BASE VERIFICATION----------------
IMPORTS TEST: (PASSED)
INITIALIZER TEST: (PASSED)
VARIABLES TEST: (PASSED)
INVARIANTS TEST: (PASSED)
increment OPERATION TEST: (PASSED)
transfer OPERATION TEST: (PASSED)
decrement OPERATION TEST: (PASSED)
FULL TEST: (PASSED)

----------------LOGIC VERIFICATION----------------
increment SAFETY TEST: (PASSED)
transfer SAFETY TEST: (PASSED)
decrement SAFETY TEST: (PASSED)

increment ABSURD TEST: (PASSED)
transfer ABSURD TEST: (PASSED)
decrement ABSURD TEST: (PASSED)

increment COMPLETENESS TEST: (PASSED)
transfer COMPLETENESS TEST: (PASSED)
decrement COMPLETENESS TEST: (PASSED)

################ CONSISTENCY TESTS ################

----------------PAIR OPPOSITION VERIFICATION----------------
increment increment OPPOSITION TEST: (PASSED)
increment decrement OPPOSITION TEST: (PASSED)
increment transfer OPPOSITION TEST: (PASSED)
decrement decrement OPPOSITION TEST: (PASSED)
decrement transfer OPPOSITION TEST: (PASSED)
transfer transfer OPPOSITION TEST: (PASSED)

----------------PAIR STABILITY VERIFICATION----------------
increment increment STABILITY TEST: (PASSED)
increment decrement STABILITY TEST: (PASSED)
increment transfer STABILITY TEST: (PASSED)
decrement decrement STABILITY TEST: (PASSED)
decrement transfer STABILITY TEST: (PASSED)
transfer transfer STABILITY TEST: (PASSED)

----------------PAIR COMMUTATIVITY VERIFICATION----------------
increment increment COMMUTATIVITY TEST: (PASSED)
increment decrement COMMUTATIVITY TEST: (PASSED)
increment transfer COMMUTATIVITY TEST: (PASSED)
decrement decrement COMMUTATIVITY TEST: (PASSED)
decrement transfer COMMUTATIVITY TEST: (PASSED)
transfer transfer COMMUTATIVITY TEST: (PASSED)

----------------EQUALITY SOLVER----------------
----------------TOKEN MODEL----------------
Tokens per operations:
```

Token conflicts:

A.5 Replicated Growable Array

Specification

```
@import;

@init
type Object;
type Ref;
type Field a;
type ListType = [Ref]<a>[Field a] a;
type Flag = [Ref]bool;
const unique content:Field Object;
const unique next:Field Ref;
const unique clock_ins:Field int;
const unique clock_up:Field int;
const tombstone:Object;
const head:Ref;
const tail:Ref;
function create_node(obj:Object, time:int) returns (result:Ref);
function get_actual_reference(list:ListType, base:Ref, time:int) returns (result:
    Ref)
{
  if (list[list[base][next]][clock_ins] < time) then (base) else (
      get_actual_reference(list, list[base][next], time))
}
function is_successor(list:ListType, base:Ref, successor:Ref) returns (result:bool
    )
{
  list[base][next] == successor || is_successor(list, list[base][next], successor)
}
axiom (forall list:ListType, presence:Flag, r:Ref :: presence[r] == presence[list[
    r][next]]);
axiom (forall list:ListType, presence:Flag :: list[tail][content] == tombstone &&
    list[tail][clock_ins] == 0 && list[tail][clock_up] == 0 && list[tail][next] ==
     tail && presence[tail] == true);
axiom (forall list:ListType, presence:Flag :: list[head][content] == tombstone &&
    list[head][clock_ins] == 0 && list[head][clock_up] == 0 && presence[head] ==
    true);
axiom (forall list:ListType, presence:Flag, r:Ref :: r!= tail && r != head &&
    presence[r] == true ==> list[r][clock_ins] > 0 && list[r][clock_up] > 0);
axiom (forall list:ListType, r1, r2:Ref :: r1 != head && r2 != head && r1 != tail
    && r2 != tail && r1 != r2 ==> list[r1][clock_up] != list[r2][clock_up] && list
    [r1][clock_ins] != list[r2][clock_ins]);
axiom (forall obj1, obj2:Object, time1, time2:int :: (obj1 != obj2 || time1 !=
    time2 ==> create_node(obj1, time1) != create_node(obj2, time2)) && (obj1 ==
    obj2 && time1 == time2 ==> create_node(obj1, time1) == create_node(obj2, time2
    )));
axiom (forall obj:Object, time:int :: create_node(obj, time) != head &&
    create_node(obj, time) != tail);
axiom (forall list:ListType, presence:Flag, r1, r2:Ref :: r1!= tail && r2 != tail
    && r1 != r2 && presence[r1] == true && presence[r2] == true ==> list[r1][next]
     != list[r2][next]);
axiom (forall list:ListType, presence:Flag, r:Ref :: presence[r] == true ==>
    is_successor(list, r, tail));
axiom (forall list:ListType, presence:Flag, r:Ref :: presence[r] == true ==>
    is_successor(list, head, r));


@variables
```

```
var list : ListType;
var presence :Flag;

@equals ListType @as forall r:Ref :: @this[r][clock_ins] == @other[r][clock_ins]
    && @this[r][clock_up] == @other[r][clock_up] && @this[r][content] == @other[r
    ][content] && @this[r][next] == @other[r][next];
@equals Flag @as forall r:Ref :: @this[r] == @other[r];

@invariant
(exists r:Ref :: presence[r] == true && list[head][next] == r) && (exists r:Ref ::
     presence[r] == true && list[r][next] == tail);

@operations

procedure delete(base: Ref, time: int)
    modifies list;
    requires (base != tail) && (presence[base] == true);
    ensures (forall r:Ref :: (r == base && old(list)[base][clock_up] < time ==>
        list[r][content] == tombstone && list[r][clock_up] == time) && (r != base
        || old(list)[base][clock_up] >= time ==> list[r][content] == old(list)[r][
        content] && list          [r][clock_up] == old(list)[r][clock_up]) && list[r
        ][next] == old(list)[r][next] && list[r][clock_ins] == old(list)[r][
        clock_ins]);


procedure update(base: Ref, obj: Object, time: int)
    modifies list;
    requires (base != tail) && (presence[base] == true);
    ensures (forall r:Ref :: (r == base && old(list)[r][clock_up] < time ==> list[
        r][content] == obj && list[r][clock_up] == time) && (r != base || old(list
        )[r][clock_up] >= time ==> list[r][content] == old(list)[r][content] &&
        list[r][clock_up          ] == old(list)[r][clock_up]) && list[r][next] ==
        old(list)[r][next] && list[r][clock_ins] == old(list)[r][clock_ins]);

procedure read(base: Ref)
    requires (base != tail) && (presence[base] == true);

procedure insert(base: Ref, new: Object, time: int)
    modifies list, presence;
    requires (base != tail) && (presence[base] == true) && (new != tombstone) && (
        presence[create_node(new, time)] == false) && (create_node(new, time) !=
        base);
    ensures (forall r:Ref :: (r == get_actual_reference(old(list), base, time) ==>
         list[r][clock_ins] == old(list)[r][clock_ins] && list[r][clock_up] == old
        (list)[r][clock_up] && list[r][next] == create_node(new, time) && list[r][
        content] == old(list)[r][content]) && (r == create_node(new, time) ==>
        list[r][clock_ins] == time && list[r][clock_up] == time && list[r][next]
        == old(list)[get_actual_reference(old(list), base, time)][next] && list[r
        ][content] == new) && (r != get_actual_reference(old(list), base, time) &&
         r != create_node(new, time) ==> list[r][clock_ins] == old(list)[r][
        clock_ins] && list[r][clock_up] == old(list)[r][clock_up] && list[r][next]
         == old(list)[r][next] && list[r][content] == old(list)[r][content])) && (
        forall r:Ref :: (r == create_node(new, time) ==> presence[r] == true) && (
        r != create_node(new, time) ==> presence[r] == old(presence)[r]));
```

Token

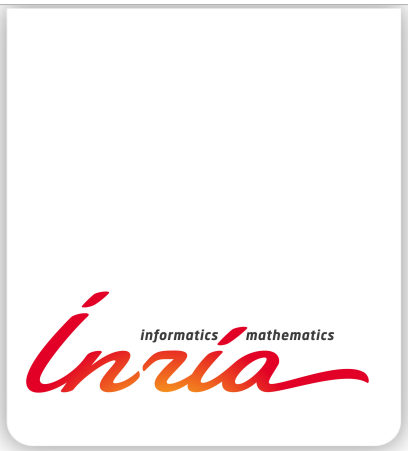
```
[
    {
        "operation1": "update",
        "operation2": "update",
        "restrictions": ["@3 != @6"]
```

```
    },
    {
        "operation1": "insert",
        "operation2": "insert",
        "restrictions": ["@3 != @6"]
    },
    {
        "operation1": "delete",
        "operation2": "update",
        "restrictions": ["@2 != @5"]
    }
]
```

Result

```
################ SPECIFICATION CORRECTION TESTS ################

----------------BASE VERIFICATION----------------
IMPORTS TEST: (PASSED)
INITIALIZER TEST: (PASSED)
VARIABLES TEST: (PASSED)
INVARIANTS TEST: (PASSED)
insert OPERATION TEST: (PASSED)
delete OPERATION TEST: (PASSED)
read OPERATION TEST: (PASSED)
update OPERATION TEST: (PASSED)
FULL TEST: (PASSED)

----------------LOGIC VERIFICATION----------------
insert SAFETY TEST: (PASSED)
delete SAFETY TEST: (PASSED)
read SAFETY TEST: (PASSED)
update SAFETY TEST: (PASSED)

insert ABSURD TEST: (PASSED)
delete ABSURD TEST: (PASSED)
read ABSURD TEST: (PASSED)
update ABSURD TEST: (PASSED)

insert COMPLETENESS TEST: (PASSED)
delete COMPLETENESS TEST: (PASSED)
read COMPLETENESS TEST: (PASSED)
update COMPLETENESS TEST: (PASSED)

----------------RESTRICTIONS VERIFICATION----------------

RESTRICTIONS WERE LOADED CORRECTLY

################ CONSISTENCY TESTS ################

----------------PAIR OPPOSITION VERIFICATION----------------
update update OPPOSITION TEST: (PASSED)
insert insert OPPOSITION TEST: (PASSED)
delete update OPPOSITION TEST: (PASSED)
delete delete OPPOSITION TEST: (PASSED)
delete read OPPOSITION TEST: (PASSED)
delete insert OPPOSITION TEST: (PASSED)
update read OPPOSITION TEST: (PASSED)
update insert OPPOSITION TEST: (PASSED)
read read OPPOSITION TEST: (PASSED)
read insert OPPOSITION TEST: (PASSED)
```

```
----------------PAIR STABILITY VERIFICATION----------------
update update STABILITY TEST: (PASSED)
insert insert STABILITY TEST: (PASSED)
delete update STABILITY TEST: (PASSED)
delete delete STABILITY TEST: (PASSED)
delete read STABILITY TEST: (PASSED)
delete insert STABILITY TEST: (PASSED)
update read STABILITY TEST: (PASSED)
update insert STABILITY TEST: (PASSED)
read read STABILITY TEST: (PASSED)
read insert STABILITY TEST: (PASSED)

----------------PAIR COMMUTATIVITY VERIFICATION----------------
update update COMMUTATIVITY TEST: (PASSED)
insert insert COMMUTATIVITY TEST: (PASSED)
delete update COMMUTATIVITY TEST: (PASSED)
delete delete COMMUTATIVITY TEST: (PASSED)
delete read COMMUTATIVITY TEST: (PASSED)
delete insert COMMUTATIVITY TEST: (PASSED)
update read COMMUTATIVITY TEST: (PASSED)
update insert COMMUTATIVITY TEST: (PASSED)
read read COMMUTATIVITY TEST: (PASSED)
read insert COMMUTATIVITY TEST: (PASSED)

----------------EQUALITY SOLVER----------------
----------------TOKEN MODEL----------------
Tokens per operations:

Token conflicts:
```



RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399